FAT Pointer based range addresses

Akilan Selvacoumar*
as251@hw.ac.uk
Heriot Watt
Edinburgh, Scotland, United Kingdom

ABSTRACT

The increasing disparity between application workloads and the capacity of Translation Lookaside Buffers (TLB) has prompted researchers to explore innovative solutions to mitigate this gap. One such approach involves leveraging physically contiguous memory to optimize TLB utilization. Concurrently, advancements in hardware-level system security, exemplified by the Capability Hardware Enhanced RISC Instructions (CHERI) architecture, offer additional opportunities for improving memory management and security.

CHERI introduces capability-based addressing, a novel approach that enhances system security by associating capabilities with memory pointers. These capabilities restrict access to memory regions, thereby fortifying the system against various security threats. Importantly, the mechanisms implemented in CHERI for enforcing memory protection can also serve as accelerators for standard user-space memory allocators. By leveraging capability-based addressing, memory allocators can efficiently manage memory resources while ensuring robust security measures are in place.

ACM Reference Format:
Akilan Selvacoumar. 2018. FAT Pointer based range addresses. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the dynamic landscape of computing, the pursuit of optimal performance is a constant endeavor, especially as applications evolve to handle increasingly complex workloads. One critical aspect influencing performance is memory management, where efficient utilization of resources is paramount. Translation Lookaside Buffers (TLBs) play a pivotal role in this regard, expediting memory access by storing recently accessed memory translations. However, as applications grow in size and complexity, the capacity of TLBs often struggles to keep pace, leading to performance bottlenecks[**?**]. To address this challenge, researchers have turned to innovative solutions, one of which involves harnessing the benefits of huge pages. Huge pages, also known as large pages, allow for the allocation of memory in significantly larger chunks compared to traditional small pages. By reducing the number of TLB entries needed to access a given amount of memory, huge pages offer a potential avenue for optimizing TLB utilization and thereby enhancing overall system performance.

Simultaneously, advancements in hardware-level security, such as the Capability Hardware Enhanced RISC Instructions (CHERI) architecture, present additional opportunities for performance enhancement. CHERI's capability-based addressing approach not only strengthens system security by tightly controlling memory access but also provides avenues for accelerating memory management operations.

In this context, the integration of huge pages into memory management strategies alongside capability-based addressing in architectures like CHERI offers a compelling synergy. By optimizing TLB utilization through the utilization of huge pages and leveraging the security features of capability-based addressing, significant performance improvements can be realized. This approach not only enhances system security but also accelerates memory access. The following below are research questions we are addressing:

(1) How does the utilization of bounds for tracking memory allocations, in addition to security purposes, affect the run times and Translation Lookaside Buffer (TLB) miss rates in modern computing systems?
(2) How does the implementation of bounds for seeking through physically contiguous memory influence the complexity and efficiency of standard memory allocators, particularly those with advanced features such as transparent huge pages, and what are the implications for system performance in terms of execution speed, memory access latency, and resource utilization?

2 FAT POINTER BASED RANGE ADDRESSES

FAT-Pointers based range addresses, combined with the capabilities of the CHERI (Capability Hardware Enhanced RISC Instructions) architecture, introduce robust memory safety and security features by incorporating additional metadata with memory pointers. This enhanced architecture utilizes concepts such as FlexPointer, Range Memory Mapping (RMM) to manage memory effectively.

Range addresses play a pivotal role within this framework, defining memory regions bounded by a starting address (Upper) and an ending address (Lower). These range addresses are encoded within FAT-pointers, allowing for precise control over memory regions.

The functionality of ranges encompasses several key aspects:

- **Creation of Physically Contiguous Memory Ranges**: By defining memory regions that are physically contiguous, systems can achieve optimal memory access patterns, enhancing performance and efficiency.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

- **Encoding Ranges as Bounds to the Pointer**: Integrating range bounds directly into FAT-pointers enables the architecture to enforce memory access restrictions at the pointer level thus allowing tracking of memory ranges on a pointer level.
- **Instrumenting Block-Based Allocators with Physically Contiguous Memory**: The integration of range-based memory concepts into memory allocation systems, such as block-based allocators, facilitates the efficient management and utilization of physically contiguous memory blocks, mitigating issues related to memory fragmentation.

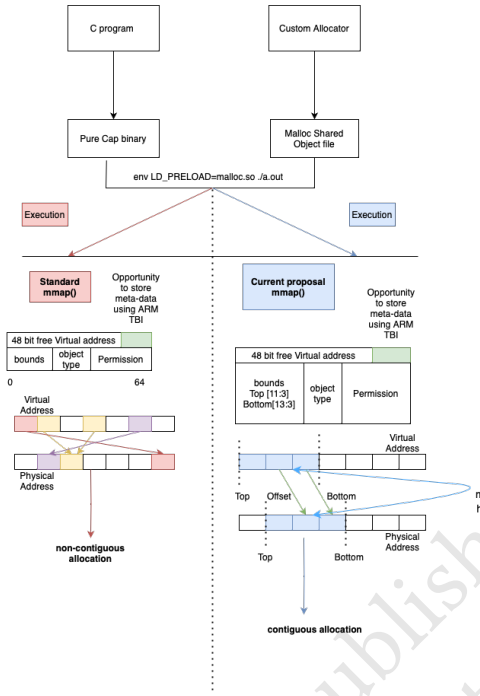

Figure 1: High overview architecture

Figure 1 illustrates the methodology employed to leverage the CHERI 128-bit FAT-pointer scheme for facilitating block-based memory management on physically contiguous memory, which is depicted on the right side of the figure. This technique contrasts with the conventional mmap approach.

In figure 1, the green-highlighted section marks the unused space between the 48th and 64th bits within the FAT-pointer. This area of unused bits presents an opportunity to store additional metadata, potentially enhancing the capabilities of the memory management system. Here we explore how this additional metadata storage could be used to further optimize memory allocation.

2.1 Range creation and huge pages

In this implementation, memory ranges are established using bounds encoded within the FAT-pointer, adhering to the CHERI 128-bit bounds compression scheme[**?**]. The memory chunk defined by the upper and lower bounds is always physically contiguous. Initially, a huge page of arbitrary size is allocated. Within this huge

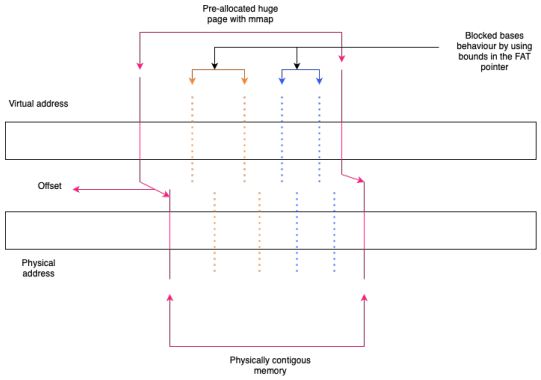

Figure 2: Range of memory

page, custom-sized memory segments are allocated using a custom-designed mmap function, which overrides the existing block-based mmap function. Once the memory is physically allocated through this custom mmap function, bounds are set to track the memory block, eliminating the need for traditional TLB usage for this purpose. Traditional TLB usage involves maintaining numerous TLB entries, often supplemented by an L2 TLB and other hierarchical structures, to translate virtual addresses to physical addresses. This approach requires multiple entries to handle various memory segments, leading to increased overhead and complexity in address translation. Conversely, the current approach streamlines this process by using a single TLB entry to translate multiple addresses within a contiguous memory range. This reduces the number of required TLB entries, simplifying the translation process and improving efficiency. By consolidating address translations into a single TLB entry, this method minimizes the overhead associated with managing numerous TLB entries and leverages the bounds encoded within the FAT-pointer for efficient memory tracking and access. This approach allows for precise and efficient memory management within the allocated huge page.

Figure 2 illustrates a straightforward use-case in which the dark pink line represents a single, large contiguous memory area, or huge page. Within this huge page, the orange and blue lines indicate two separate memory allocations equivalent to invoking malloc twice to allocate memory in distinct regions. This scenario simulates a block-based memory allocator operating within the confines of the huge page. The allocations leverage the bounds encoded in the FAT-pointer, ensuring tracking and efficient management of the allocated memory regions. By using the FAT-pointer bounds, this method maintains the integrity and contiguity of the allocated blocks within the huge page.

2.2 Software Stack

The software stack is based on CHERIBSD, selected because ARM officially supports Morello's performance counters on this operating system. As illustrated in the figure **??**, the setup includes a C program that is linked to the prototype memory allocator or to various memory allocators being benchmarked. This linkage can occur in two ways: either as a shared object file during compile time for larger allocators, or as a header file for smaller allocators,

ensuring flexibility and efficiency in memory management.

This integration ensures that the memory allocation process is optimized for performance, leveraging the contiguity of memory blocks and the capabilities provided by the CHERI architecture and the Morello platform. By using the contigmem driver and the custom mmap function, the system achieves efficient memory allocation and tracking, crucial for the high-performance needs of the application.

2.3 Contigmem driver from DPDK

The custom mmap function, tailored to ensure physically contiguous memory allocation, is a key component of this system. This function is linked to the contigmem driver, which has been modified from the DPDK[?] library to meet the specific needs of this implementation. The contigmem driver is essential for managing large contiguous memory blocks and is loaded during the system boot process. It reserves a huge page of arbitrary size, with the size parameter set based on the requirements of the conducted experiments.

Listing 1: Contigmem driver

```
MALLOC_DEFINE(M_CONTIGMEM, "contigmem",
"contigmem(4) allocations");

static int contigmem_modevent(module_t mod,
int type, void *arg)
{
  int error = 0;

  switch (type) {
  case MOD_LOAD:
    error = contigmem_load();
    break;
  case MOD_UNLOAD:
    error = contigmem_unload();
    break;
  default:
    break;
  }

  return error;
}

....

DECLARE_MODULE(contigmem, contigmem_mod,
SI_SUB_DRIVERS, SI_ORDER_ANY);
MODULE_VERSION(contigmem, 1);

static struct cdevsw contigmem_ops = {
        .d_name = "contigmem",
        .d_version = D_VERSION,
        .d_flags = D_TRACKCLOSE,
        .d_mmap_single = contigmem_mmap_single,
        .d_open = contigmem_open,
        .d_close = contigmem_close,
};

static int
contigmem_load()
{
        ....

        for (i = 0; i < contigmem_num_buffers; i++) {
                addr = contigmalloc(contigmem_buffer_size,
          M_CONTIGMEM, M_ZERO,
                  0, BUS_SPACE_MAXADDR,
          contigmem_buffer_size, 0);
        ....
        }

    ....

error:
        for (i = 0; i < contigmem_num_buffers; i++) {
                if (contigmem_buffers[i].addr != NULL) {
                        contigfree(contigmem_buffers[i].addr,
                                contigmem_buffer_size, M_CONTIGMEM);
                        contigmem_buffers[i].addr = NULL;
                }
                ....
        }

        return error;
}
```

When the contigmem_load function is called, either during boot or when the Kernel module is loaded, it pre-allocates a segment of physically contiguous memory. This approach differs from FlexPointer[?], which allocates physically contiguous memory eagerly. The contigmem_load function allocates memory using contigmalloc, which allocates physically contiguous memory initialized to zero. The cdevsw struct refers to function calls which would be overwritten on loading the driver. In the code snippet above the mmap function would be overwritten with contigmem_mmap_single if the following driver is opened and truncated as shown in Code snippet.

In the code snippet the cdev_pager_ops refers to the operations which will be overwritten when called with mmap such as overwriting page faults.

Listing 2: Contigmem driver mmap

```
#define FILENAME "/dev/contigmem"
void *ptr;
int MallocCounter;

size_t sizeUsed;

...

INITAlloc(void) {
```

```
  size_t sz;
  sz = 100000000;

  int fd = open(FILENAME, O_RDWR, 0600);

   if (fd < 0) {
       perror("open");
       exit(EXIT_FAILURE);
   }

   off_t offset = 0; // offset to seek to.

   if (ftruncate(fd, sz) < 0) {
       perror("ftruncate");
       close(fd);
       exit(EXIT_FAILURE);
   }

   ptr = mmap(NULL, sz,
   PROT_READ|PROT_WRITE, MAP_SHARED,fd,0);

  // Added error handling
   if(ptr == MAP_FAILED)
   {
       perror("mmap");
       exit(EXIT_FAILURE);
   }
   MallocCounter = (int)sz;
   }

   void* malloc(size_t sz)
   {
  sz = __builtin_align_up(sz, _Alignof(max_align_t));

  // printf("%d \n", sz);
  // printf("%d Malloc counter\n", MallocCounter);

  MallocCounter -= sz;
  void *ptrLink = &ptr[MallocCounter];
  ptrLink = cheri_setbounds(ptrLink, sz);

  return ptrLink;

  }

  void FREECHERI(void *ptr) {

  // get length of free from bounds
  // in the pointer
  int len = cheri_getlen(ptr);

  munmap(ptr, len);
}
```

The code snippet.; below shows a sample memory allocator which is a really simple implementation that initially in the example allocates 1GB of memory using the mmap call which calls the mmap function from /dev/contigmem driver. This ensures memory allocated to the physically contiguous memory allocated in the contigmem_load() function using the contigmem_mmap_single() function call in the kernel module, uses malloc and free to allocate within this memory chuck. The consideration of this is to ensure that a C program needs minor changes to use the benefit using physically contiguous memory with bounds within a segment of memory.

3 RELATED WORK

3.1 Huge Pages

Increasing TLB reach can be achieved by using larger page sizes, such as huge pages[**?**], which are common in modern computer systems. The x86-64 architecture supports huge pages of 2 MB and 1 GB, backed by OS mechanisms like Transparent Huge Pages (THP) and HugeTLBFS in Linux. However, available page sizes in x86-64 are limited, leading to internal fragmentation issues. For instance, allocating 1 MB with 4 KB base pages requires 256 PTEs, but using a 2 MB huge page would waste half of the memory space. Some architectures offer more page size choices, such as Intel Itanium, which allows different areas of the address space to have their own page sizes. Itanium uses a hash page table to organize huge pages, but without significant changes to the conventional page table, it only helps reduce page walk overheads. HP Tunable Base Page Size permits the OS to adjust the base page size, but still faces internal fragmentation problems, with HP recommending a base page size of no more than 16 KB. Shadow Superpage introduces a new translation level in the memory controller to merge non-contiguous physical pages into a huge page in a shadow memory space, extending TLB coverage. However, this approach requires all memory traffic to be translated again in the memory controller, resulting in additional latency for memory accesses.

3.2 Segment

Early processors often used segments to manage virtual memory, where a segment essentially mapped contiguous virtual memory to contiguous physical memory. Unlike pages, which are relatively small, segments can be much larger, offering the potential for more efficient memory management in certain scenarios. This concept of segmentation has seen a resurgence in some modern approaches that aim to enhance translation coverage by designating specific areas in the virtual address space.

This method allows programmers to explicitly define a single segment for applications requiring significant memory. It introduces two new registers to the system, which indicate the start and end of this segment. Virtual addresses within this segment are translated by calculating the offset from the virtual start address and applying this offset to the physical start address. This straightforward method simplifies the translation process for large memory areas but requires significant modifications to the source code of applications.

3.3 Range Memory Mapping (RMM):

Redundant Memory Mappings (RMM)[**?**] enhance memory management by introducing an additional range table that pre-allocates contiguous physical pages for large memory allocations, creating ranges that are both virtually and physically contiguous. This approach simplifies address translation within these ranges by adding an offset, similar to Direct Segment, but RMM supports multiple ranges and operates transparently to programmers, requiring no source code modifications. The range table, separate from the conventional page table, holds the mappings for these large allocations. To determine which range an address belongs to, RMM compares the address against all range boundaries, a process that is computationally expensive and therefore performed only after an L1 TLB miss. To optimize this, RMM uses a range TLB (RTLB) to quickly identify if an address falls within any pre-allocated range, facilitating efficient translation and reducing overhead. Range mapping works alongside the paging system by generating TLB entries on TLB misses and still performing TLB lookups for each virtual address translation. Unlike traditional segmentation mechanisms, range mapping activates a range lookaside buffer (RTLB) located with the last level TLB upon a miss. The hardware TLB miss handler then searches the RTLB for the miss address and, if found, generates a new TLB entry with the physical address derived from the base virtual address and range offset, along with permission bits. If the RTLB also misses, the system defaults to a standard page walk while a range table walker simultaneously loads the range into the RTLB in the background, avoiding delays in memory operations. The RTLB, functioning as a fully associative search structure, ensures that most last level TLB misses are handled efficiently by range mapping, reducing the need for costly page table walks.

3.4 CHERI

CHERI (Capability Hardware Enhanced RISC Instructions) extends conventional processor Instruction-Set Architectures (ISAs) with architectural capabilities to enable fine-grained memory protection and highly scalable software compartmentalization. CHERI is a hybrid capability architecture that can combine capabilities with conventional MMU(i.e Memory Management Unit) based systems. The contribution of the following project include:

- ISA changes to introduce architecture capabilities.
- New microarchitecture proving that capabilities can be implemented efficiently in hardware. Support for efficient tagged memory to protect capabilities and compress capabilities to reduce memory overhead.
- Newly designed software construction model for that uses capability to provide fine grain memory protection and scalable software compartmentalization.
- Language and Compiler extension to use capabilities for C and C++.
- OS extensions to use (and support application use of) fine-grained memory protection (spatial, referential, and (non-stack) temporal memory safety) and abstraction extensions to support scalable software compartmentalization.

3.5 Future work

The current experimental setup on the ARM Morello board is constrained by the requirement that all memory reads must pass through the Translation Lookaside Buffer (TLB) for address translation. This necessitates frequent TLB lookups, potentially leading to performance bottlenecks. The planned future work aims to address this by leveraging CHERI (Capability Hardware Enhanced RISC Instructions) extensions on the RISC-V architecture, specifically using the Tooba implementation.

3.5.1 Storing Offsets Directly on Pointers. In the current ARM Morello setup, address translations rely on the TLB. The future approach on RISC-V Tooba involves storing the offset directly within the pointer. This is possible due to CHERI's capability model, which supports fine-grained memory protection and can encode bounds within pointers. Utilizing Bounds in CHERI for Block-Based Allocation:

CHERI capabilities allow pointers to carry metadata about memory bounds, providing hardware-enforced memory safety. By encoding the offset and bounds within the pointer, the system can directly access memory without needing intermediate translations via the TLB. This enables the implementation of a block-based allocator that can efficiently manage memory allocations and deallocations within defined bounds. Bypassing the TLB in RISC-V Tooba.

3.5.2 Hardware Modifications: The Bluespec design of the RISC-V processor will be modified to allow certain memory operations to bypass the TLB. This means that when a pointer with encoded offset and bounds is used, the system can directly compute the physical address from the capability information. This modification reduces the dependency on the TLB, decreasing latency and improving performance, especially for frequent memory operations. Transition to a Single-Address-Space Operating System (SASOS)[**?**].

3.5.3 Concept of SASOS:. In traditional operating systems, there is a clear separation between user space and kernel space. This separation is enforced by memory protection mechanisms and address translation through the TLB. In a Single-Address-Space Operating System, this distinction is removed. Both user applications and the kernel share the same contiguous address space.

3.5.4 Advantages of SASOS with CHERI:.

- Simplified Memory Management : Without the need to switch between user and kernel spaces, memory management becomes simpler and more efficient. The kernel allocator can be the same as the user space allocator, operating on a single, contiguous chunk of memory.
- Unified Allocator: The unified memory allocator can efficiently manage memory for both kernel and user applications, leveraging CHERI's capability-based protection to prevent unauthorized access. This reduces overhead and potential fragmentation issues associated with maintaining separate memory spaces.

4 CONCLUSION

This paper addresses the growing disparity between application workloads and the capacity of Translation Lookaside Buffers (TLBs).

To mitigate this gap, it proposes leveraging physically contiguous memory to optimize TLB utilization. Additionally, the report explores advancements in system security, particularly through the Capability Hardware Enhanced RISC Instructions (CHERI) architecture. CHERI's capability-based addressing enhances system security by associating capabilities with memory pointers, restricting access to memory regions, and thus protecting against various security threats. Importantly, these mechanisms can also improve the efficiency of memory allocators by managing memory resources while ensuring robust security measures.

This paper highlights the constant pursuit of optimal performance in computing, emphasizing the importance of efficient memory management. TLBs are crucial in expediting memory access by storing recently accessed memory translations. However, as applications grow in size and complexity, TLB capacity often becomes a bottleneck. One innovative solution is the use of huge pages, which allocate memory in larger chunks, thereby reducing the number of TLB entries required and potentially enhancing overall system performance. Advancements in hardware-level security, such as CHERI's capability-based addressing, offer additional performance enhancement opportunities by tightly controlling memory access and accelerating memory management operations. Integrating huge pages into memory management strategies alongside CHERI's capability-based addressing can optimize TLB utilization and leverage security features for significant performance improvements.

This paper aims to demonstrate how leveraging physically contiguous memory and advanced security architectures like CHERI can enhance memory management efficiency while ensuring robust security measures. These advancements ultimately contribute to improved system performance, addressing the challenges posed by the increasing complexity and size of modern application workloads.

REFERENCES

[6]]bidpdk-based2016 Hao Bi and Zhao-Hun Wang. [n. d.]. DPDK-based Improvement of Packet Forwarding. 7 ([n. d.]), 01009. https://doi.org/10.1051/itmconf/20160701009 Publisher: EDP Sciences.

[6]]chenflexpointer2023 Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. [n. d.]. FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers. 20, 2 ([n. d.]), 1–24. https://doi.org/10.1145/3579854

[6]]esswoodcheriosnodate Lawrence G Esswood. [n. d.]. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor. ([n. d.]).

[6]]karakostasredundant2015 Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. [n. d.]. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland Oregon, 2015-06-13). ACM, 66–78. https://doi.org/10.1145/2749469.2749471

[6]]panwarhawkeye2019 Ashish Panwar, Sorav Bansal, and K. Gopinath. [n. d.]. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence RI USA, 2019-04-04). ACM, 347–360. https://doi.org/10.1145/3297858.3304064

[6]]woodruffcheri2019 Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. [n. d.]. CHERI Concentrate: Practical Compressed Capabilities. 68, 10 ([n. d.]), 1455–1469. https://doi.org/10.1109/TC.2019.2914037